第4章 センサ・データの取得からノード間の送受信までをプログラミング

センサ・ネット専用OSと拡張C言語による開発事例

ワイヤレス・センサ・ネットワーク編

宮本 哲

ここでは，センサ・ネットワークのソフトウェアの開発事例を紹介する．センサのデータを取得し，ノード間の送受信を行うためのプログラミングを取り上げる．センサ・ネットワークに特化して開発されたOSと拡張C言語を使用する．なお，ここで紹介したソース・コードは，本誌付属のCD-ROMに収録されている．（編集部）

ワイヤレス・センサ・ネットワークは，センサ・ノードを自由に配置できるという特徴を持ちます．あらゆる場所に簡単に配置できるようにセンサ・ノードは小型でなければなりません．また，ほとんどの場合，センサ・ノードは電池で駆動することが求められます．これらの要件から，センサ・ノードに使われるマイクロプロセッサや無線通信モジュール，センサは小型で低消費電力のものでなければなりません．

センサ・ノードが低消費電力で小型であるということは，マイクロプロセッサの性能は低く，メモリなどの資源も非常に限られるということです．低消費電力処理に加えて，ソフトウェア開発ではこういった制限も考慮しなければなりません．具体的には，以下のようなことがソフトウェアに求められます．

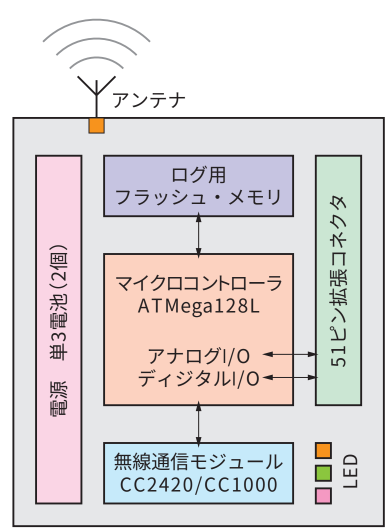
- プログラム・サイズが小さい
- メモリを消費しない
- むだな処理を避ける
- マイクロプロセッサをなるべくスリープさせる．無線通信モジュールやセンサは使わないときにOFFにする

現時点では，ワイヤレス・センサ・ネットワークの応用分野として考えられているものは多岐にわたり，一つのマスマーケットが見つかっているわけではありません．したがって，さまざまな用途に対応できるようにセンサ・ノードにつけるセンサは取り替え可能であり，センサ・ノードのプログラムは変更が容易であることが望ましいといえます．

本稿では，これらの要件を満たすワイヤレス・センサ・ネットワークの開発事例を紹介します．具体的には，米国Crossbow Technology社のセンサ・ネットワーク向け無線モジュール「MOTE」とセンサ・ネットワーク用OS「TinyOS」を例に挙げて，おもにソフトウェア開発について解説します．



（a）外観（MICAz）



（b）構成図

図1
本稿で使用する無線モジュール
（a）は，IEEE 802.15.4準拠の無線モジュール「MOTE MICAz」の外観である．（b）に構成を示す．51ピン拡張コネクタを介してセンサ・ボードを接続する．また，ログ用のフラッシュ・メモリを備えている．電源については単3電池2個で駆動する．

1 センサ・ネットワークの構成要素

最初に，本稿で取り上げるセンサ・ネットワーク・システムを構築するためのハードウェア，ソフトウェアについて説明します．

● ハードウェア――無線モジュールとセンサ・ボード

図1に，無線モジュール「MOTE」の概要を示します(1), (2)．本無線モジュール単体ではセンサを備えていません．拡張用の51ピン・コネクタを介して，さまざまなセンサ・ボードを接続して使います．

プログラムを書き込んだり，センサのデータをパソコンへ送信するには，インターフェース・ボードを用います．このインターフェース・ボードはシリアル・インターフェースを備えており，これをパソコンにつないで使います．このほかEthernetを備えたボードもあります．

主要なハードウェアとしては，上記のもの以外にゲートウェイ(Stargate)があります．これは，CPUに米国Intel社のXScaleを使い，その上でLinuxを動作させています．そのほか，シリアル・ポート，カード・インターフェース(PCMCIAやCompactFlash)，Ethernetポート，USBポート，拡張用51ピン・コネクタを備えています．ただし，このゲートウェイは電池駆動ではありません．

● ソフトウェア――センサ・ネットのための専用OS

無線モジュールに搭載されているソフトウェアは「TinyOS」と呼ばれるOSとそのアプリケーション・ソフトウェアです．

TinyOSはワイヤレス・センサ・ネットワーク用に開発されたOSです(3)．本OSはオープン・ソースであり，多くの開発者によって機能の追加や改良が行われています．情報交換やサポートを行うためのメーリング・リスト(http://www.tinyos.net/special/support/)や，今後の機能拡張について議論するワーキング・グループもあります．

本OSは，先に述べたワイヤレス・センサ・ネットワークのソフトウェアに求められる条件，すなわち「軽量であること」を満たすために開発されました注1．したがって，一般的なOSとは異なり，プリエンプティブ・マルチタスク(複数のタスクを強制的に切り替えるしくみ)やメモリ保護機能といったサービスは提供しません(図2)．本OSは，OSというよりもハードウェア制御やネットワーク・プロトコル・スタックなどのソフトウェア・ライブラリ注2から構成されるフレームワーク注3といったほうが正確です．このフレームワークにより，限られた資源を使って高い性能を実現することと，ソフトウェアの開発効率を上げることのバランスをうまくとっています．

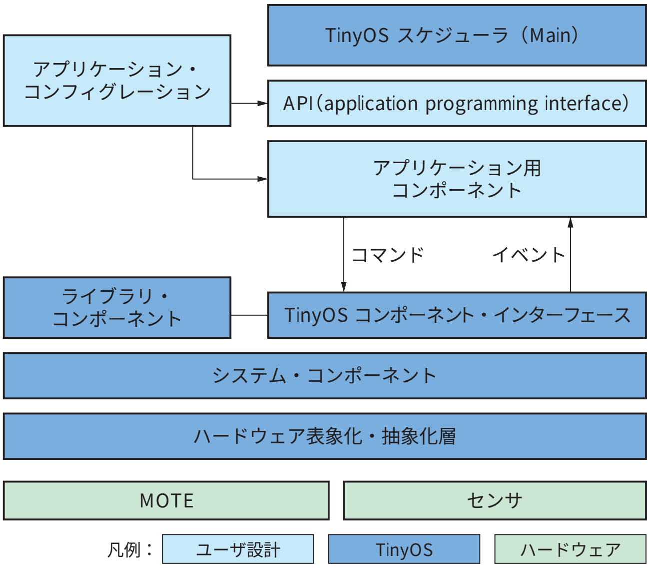


図2 TinyOSモジュール構成

TinyOSはハードウェア(MOTEやセンサ・ボード)から上へ階層構造をとっているため，モジュールの差し替えや追加が容易である．アプリケーション開発者(ユーザ)は，上位層のTinyOSコンポーネント・インターフェースを使って効率良くアプリケーション・ソフトウェアを開発できる．

● 拡張C言語によるプログラミング

TinyOSの根幹をなしているのがnesCというC言語を拡張したTinyOS用のプログラミング言語注4です．nesCにはTinyOS向けのプログラミングに対応するため，いくつかのC言語拡張が含まれています．おもな拡張としては，コンポーネント・プログラミングのための拡張(ワイヤリング)とタスク処理のための拡張の二つが挙げられます．

1) コンポーネント・プログラミング

nesCではコンポーネント(component)と呼ぶソフトウ

注1：OSなしの場合と比べてOSを実装している無線モジュール(MOTEなど)のほうが処理が重くなると指摘されることがあるが，一概にそうとは言えない．
注2：本稿では説明しないが，OAP(Over the Air Programming)というワイヤレス・ネットワークを介してセンサ・ノードのプログラムを書き換えるモジュールもある．
注3：ソフトウェア・システムを設計，実装するための枠組み．
注4：nesCのリファレンス・マニュアルは「http://www.tinyos.net/tinyos-1.x/doc/nesc/ref.pdf」から入手できる．本稿執筆時点(2005年7月)では日本語版を校正中であり，これが終わりしだいクロスボーのホームページ(http://www.smartdust.jp/)にて公開する予定．

ェア・モジュールを組み上げてプログラムを作ります．このコンポーネントはインターフェース（interface）を提供（provide）したり，あるいはほかのコンポーネントによって提供されるインターフェースを利用（use）したりすることができます．インターフェースの提供と利用を介してコンポーネントは結び付けられます．

インターフェースは，コマンド（command）とイベント（event）から構成されます．コマンドは機能を表し，イベントは事象の発生を表します．インターフェースを利用するコンポーネントはcall文でコマンドを呼び出してその機能を使います．一方，インターフェースを提供するコンポーネントはsignal文でイベントを発行し，その事象の発生を通知します．コマンドはそのインターフェースを提供するコンポーネントが実装し，イベントはそのインターフェースを利用するコンポーネントが実装します．

このようにインターフェースを挟んで提供側と利用側がコマンドとイベントにより，それぞれの方向へ結ばれていることから，nesCのインターフェースは双方向性を持つと言います．この関係を**図3**に示します。

より理解を深めるために，タイマ・インターフェースを提供するタイマ・コンポーネントを例に説明しましょう．

タイマ・インターフェースは，指定した時間が経過したときになんらかの処理を行いたい場合に使います．タイマ・インターフェースを利用するコンポーネントは，タイマを開始するためにタイマ・インターフェースのstartコマンドを呼び出します．指定された時間が経過すると，タイマ・コンポーネントはこれを通知するため，firedイベントを発行します．この例では，タイマ・インターフェースを提供するタイマ・コンポーネントがstartコマンドを実装し，タイマ・インターフェースを利用するコンポーネントがfiredイベントを実装します[注5]．

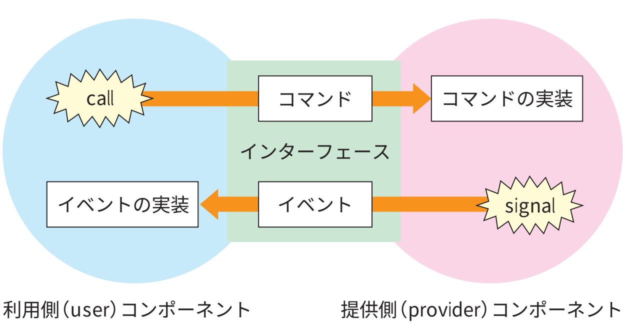


図3　nesCのインターフェースとコンポーネント

利用側コンポーネントはインターフェースのコマンドをcallで呼び出して，提供側のコマンドの実装を呼び出す．提供側コンポーネントはインターフェースのイベントをsignalで発行して，利用側のイベントの実装を呼び出す．

nesCのもっとも基本的なプログラミング・モデルは，必要なコンポーネントのコマンドを呼び出し，必要なイベント処理を実装していくことになります．TinyOSは一般的なGUIプログラミングと同様のイベント駆動型のアプリケーション・フレームワークを提供しているのです．

コンポーネントにはコンフィグレーション（configuration）とモジュール（module）の2種類があります．モジュールはそれ自身のインスタンス構造の定義を持ち，提供するインターフェースのコマンドや利用するインターフェースのイベントの実装を含みます．

一方，コンフィグレーションは自身のインスタンスは持たない代わりに必要なコンポーネントのインスタンス宣言を含み，これらのインターフェースをほかのコンポーネントのインターフェースへとワイヤリング（提供・利用関係にある二つのコンポーネントを=，->構文によって結ぶこと）します．ワイヤリングに関しては実際のソース・コードを見ながら説明したほうがわかりやすいと思うので，後ほど詳しく述べます．

nesCのコンパイラは，実際にはプリプロセス（前処理）を行ってCプログラムに変換した後，gccでコンパイルを行っています．

2）タスク・モデル

TinyOSにはメモリ空間の独立した「プロセス」[注6]という概念は存在しません．一つのアプリケーションのみが実行され，この実行オブジェクトに本OSのすべてが静的にリンクされています．

代わりに，本OSではタスクと呼ぶ並列処理の実行単位が用意されています．タスクということばを一般的にとらえてしまうと非常にあいまいになります．本OSでいうタスクは，メモリ共有型のプロセス（あるいはライト・ウェイト・プロセスと呼ばれるもの）に近いものです．nesCの記憶クラス指定子taskを付けて宣言された関数がタスク

注5：nesCをC++やJavaなどのオブジェクト指向言語にたとえて説明すると，コンポーネントはクラス（オブジェクト指向プログラミングにおいて，データとそれを操作する手続きをまとめたオブジェクト定義）に，コマンドやイベントはメソッド（クラスのデータを操作する手続き）に該当する．また，イベントは仮想関数であると言える．イベントを利用側が実装するのは，派生クラス（継承によって作られたクラス．元のクラスを基底クラスと呼ぶ）が仮想関数をオーバライドすること（基底クラスのメソッドのうち，仮想関数として定義されているものを派生クラスで再定義すること）をイメージすれば理解しやすいだろう．

注6：OSが実行できる処理単位．

の実体であり，この関数ポインタがシステムのタスク・キューへ登録され，順番に実行されていきます．タスク・キューはFIFO（first-in first-out）であり，登録された順番にしたがってキューから取り出されます．

並列処理といっても，本OSはプリエンプティブなマルチタスクのサービスを提供していないので，明示的に行わない限り[注7]，OSのタスクが互いに割り込むことはありません．

ただし，本OSにも割り込みハンドラは存在します．これはハードウェア割り込みによって呼び出される処理であり，OSのタスクに割り込むことができます．タスク・キューが空の状態では，ハードウェア割り込みが発生するまでスリープし続けます．つまり，処理すべきタスクがない間はつねにスリープ状態になっています．タスクと割り込みハンドラの関係を**図4**に示します．

2 アプリケーション開発の実際

以上を踏まえて，TinyOSアプリケーションのプログラミングを実際のソース・コードをもとに説明していきます．なお，ここではTinyOSをインストールした環境を前提として話を進めます．TinyOSそのものは，米国University of California，Barkeley（UC Barkeley）の本OSに関するホームページ（http://www.tinyos.net/）からダウンロードできます．その後，Crossbow社のサポート用Webサイト（http://www.xbow.com/Support/support.htm）から必要なソフトウェアをダウンロードします．これらのソフトウェアのインストールや実行ファイルをビルドする方法については，参考文献(4)，(5)を参照してください[注8]．

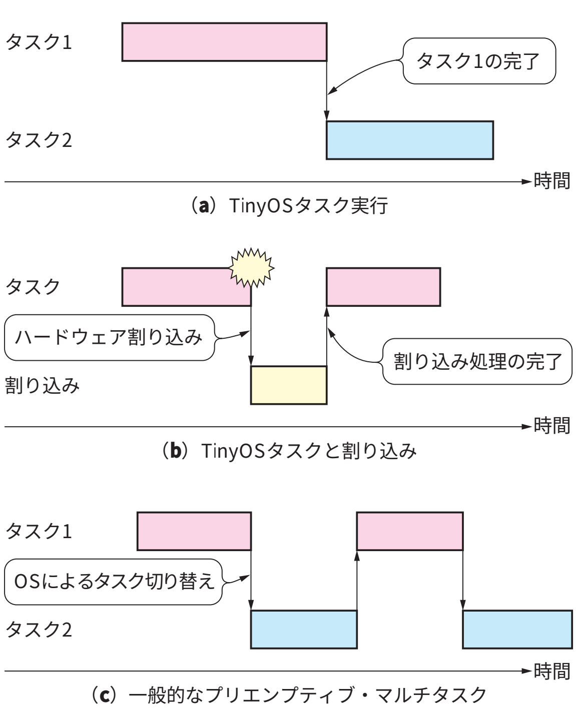


図4　TinyOSにおけるタスクと割り込み

(a)では，タスクどうしで割り込むことはなく，かならずあるタスクが完了してから別のタスクが実行される．(b)では，ハードウェア割り込みがタスクに割り込み，割り込み処理が実行される．(c)は参考として，一般的なプリエンプティブ・マルチタスクの処理を示したもの．この場合，タスクの実行途中でOSによって別のタスクへ処理が切り替わる．

● まずはLED点滅のプログラミングを考えてみる

アプリケーションとして，やはり最初は“Hello World”を表示することから始めたいところですが，今回使用する無線モジュール（MOTE）には表示デバイスがないため，文字を表示できません．代わりにLEDを光らせるアプリケーションから始めることにしましょう．

リスト1がMOTE版“Hello World”アプリケーションである`MyFirstApp`のソースです．通常のC言語のプログラムが一つの`main`関数で表されるように，TinyOS/nesCのアプリケーションは`Main`コンポーネントを含んだ一つのコンフィグレーション・コンポーネントで表されます．MyFirstApp.ncの`MyFirstApp`というコンフィグレーションがこれに該当します．

nesCではコンポーネントの名まえはファイル名と同じでなければなりません．nesCはコンポーネント名と同じ名ま

リスト1　LED点滅用トップレベルの記述（MyFirstApp）

nesCに対応したコンパイラなどのツールは日本語の利用を保証しているわけではないが，本稿ではわかりやすくするためにリストに日本語のコメントを用いている．

```
configuration MyFirstApp {
    // 規定部
}
implementation {
    // 実装部
    components Main,TimerC,LedsC,MyFirstAppM;

    Main.StdControl -> TimerC;
    Main.StdControl -> MyFirstAppM;
    MyFirstAppM.Leds -> LedsC;
    MyFirstAppM.Timer -> TimerC.Timer[unique("Timer")];
}
```

注7：`signal`文によるイベントの通知を指しているが，厳密にはこれもタスクの割り込みとは言えない．

注8：Crossbow社のセンサ・ネットワーク向け無線モジュール（MOTEシリーズ）を購入している場合は，その製品に付属するCD-ROMにしたがってインストールする．

4

えのファイルをソース探索パス上から見つけ出し，これを読み込んで処理します．TinyOS内のコンポーネントのソースを探す場合は，tosディレクトリの下のsystem，platform，libといったサブディレクトリ内で目的のコンポーネント名のファイルを探してください．

configuration識別子に導かれたブロックは規定部（specification）と呼ばれ，ここに提供側と利用側のインターフェースを列挙しますが，MyFirstAppはトップレベルのコンポーネントなのでいずれもありません．

implementation識別子に導かれるブロックは実装部（implementation）と呼ばれ，コンフィグレーションが使うコンポーネントを宣言し，それらのインターフェースやコマンド，イベントをワイヤリングします（**表1**，**表2**）．

●「ワイヤリング」と「インターフェース」がポイント

MyFirstAppはアプリケーション・コンフィグレーションなので，かならずMainコンポーネントを含んでいます．Mainコンポーネント（tos/system/Main.nc）はStdControlインターフェースを利用します．StdControlインターフェース（tos/interfaces/StdControl.nc）はinit()，start()，stop()という三つのコマンドを持っています．これらはそれぞれ初期化，開始，終了に相当するコマンドです．Mainコンポーネントはアプリケーションの初期化，開始，終了のタイミングで，それぞれのコマンドを呼び出します．したがって，アプリケーションと同じタイミングでコンポーネントの初期化や開始，終了といった処理を行う必要があるものは，StdControlインターフェースを提供し，これをMainへワイヤリングしなければなりません．

ここで，MainのStdControlインターフェースをTimerCコンポーネントとMyFirstAppMコンポーネントにワイヤリングしています．これでTimerCとMyFirstAppMの初期化，開始，終了処理をアプリケーション実行時の適切なタイミングで行うことができるのです．

TimerCはタイマでLEDを光らせるために使います．TimerCはTimer[uint8_t id]インターフェースとStdControlインターフェースを提供するコンポーネントです．Timer[uint8_t id]インターフェースはパラメータ付きインターフェースです．

表1　MyFirstAppで使うインターフェースのおもなコマンドとイベント

StdControl	標準コンポーネント・インターフェース	
コマンド	init	コンポーネントの初期化を行う
	start	コンポーネントを開始する
	stop	コンポーネントを停止する
Timer	タイマを利用するためのインターフェース	
コマンド	start	タイマを開始する
	stop	タイマを停止する
イベント	fired	指定時間に達したときに通知される
Leds	LEDを利用するためのインターフェース	
コマンド	init	LEDを初期化する
	redOn	赤のLEDを点灯する
	redOff	赤のLEDを消灯する
	redToggle	赤のLEDの状態を反転させる
	set	0～7までの値を赤・緑・黄のLEDで表現する
ADC	A-Dコンバータを利用するためのインターフェース	
コマンド	getData	A-D変換を開始する
イベント	dataReady	A-D変換完了時に通知される
Send	パケットを送信するためのインターフェース	
コマンド	send	パケットを送信する
	getBuffer	ペイロードを格納するパケット中の場所を返す
イベント	sendDone	パケット送信が完了したときに通知される

表2　MyFirstAppで使うコンポーネントのおもな提供・利用インターフェース

Main	アプリケーションのメイン処理を行う
利用インターフェース	StdControl
TimerC	タイマ処理を行う
提供インターフェース	Timer
	StdControl
LedsC	LEDの制御を行う
提供インターフェース	Leds
PhotoTempM	光・温度センサの制御を行う
提供インターフェース	StdControl as TempStdControl
	StdControl as PhotoStdControl
	ADC as ExternalTempADC
	ADC as ExternalPhotoADC
利用インターフェース	ADCControl
	ADC as InternalTempADC
	ADC as InternalPhotoADC
	StdControl as TimerControl
	Timer as PhotoTempTimer
EWMAMulti HopRouter	XMeshネットワーク処理を行う
提供インターフェース	StdControl
	Send
	Receive
	Intercept
	Intercept as Snoop
	RouteControl
利用インターフェース	ReceiveMsg
Generic CommPromiscuous	低レベルの送受信処理を行う
提供インターフェース	StdControl as Control
	CommControl
	SendMsg
	ReceiveMsg
利用インターフェース	sendDone()

パラメータ付きインターフェースとは，このインターフェースの利用が複数あって，かつそれらが別々のインスタンスを使う必要がある場合に用います．つまり，パラメータ idによって利用者を区別できるわけです[注9]．unique関数は，同一の引き数（ここではTimer）で呼び出すたびにユニークな番号を返す定数関数[注10]です．このユニークな番号によって複数のTimerインターフェースを区別しています．MyFirstAppMはTimerインターフェースを利用するので，ここへワイヤリングします（下掲のコラム「ワイヤリングの意味」を参照）．

LedsCはLEDを制御するコンポーネントです．LEDを制御するためのLedsインターフェースを提供しており，このインターフェースを利用するMyFirstAppMにワイヤリングされています（**図5**）．また，ワイヤリングの図は

注9：TinyOSのコンポーネントは，たとえ複数箇所から利用されていても，原則として一つのインスタンスしか持たない．
注10：コンパイル時に値が決定されるということ．

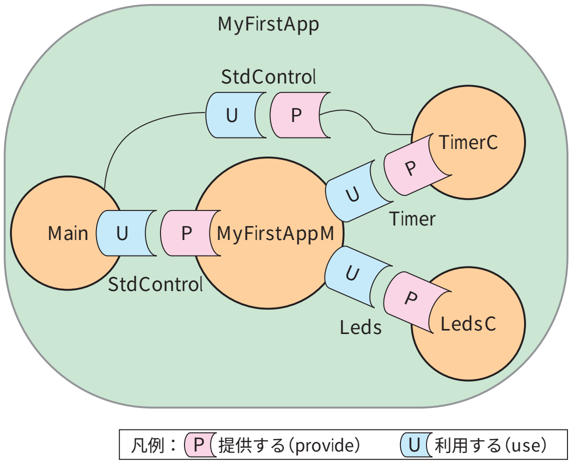
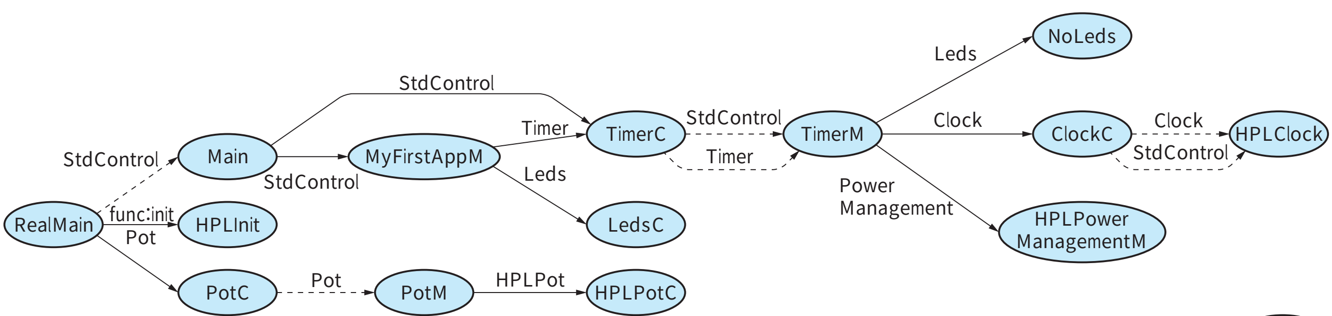
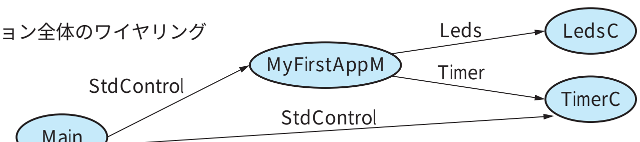


図5　MyFirstAppのワイヤリング
ワイヤリングの基本をわかりやすく説明するため，提供側と利用側のインターフェースが互いにつながっているようすを図にした．外側に対して凹凸いずれかで利用と提供を区別している．実際にはコンポーネント構成によっては内部で利用インターフェースどうしをつなぐこともあるためこのような形にしたが，本稿ではそこまでは説明しない．



（a）MyFirstAppアプリケーション全体のワイヤリング



（b）MyFirstAppでのワイヤリング

図6
MyFirstAppのワイヤリング図
nesCのドキュメント生成機能を用いると，図のようなワイヤリング図が自動生成される（ただし，本図はイメージ図）．アプリケーションから直接見えない階層でもさまざまにワイヤリングが行われていることがわかる．

Column　ワイヤリングの意味

TinyOS/nesCにおけるワイヤリングとは，インターフェースの提供・利用の関係にある二つのコンポーネントを結ぶことを指します（場面によっては提供どうし，利用どうしで結ぶこともある）．nesCのC言語拡張文法により=や->といった記法を用いてコンポーネントを文字どおり「線で結んでいく」感覚で組み合わせることができます．

では，ワイヤリングとはいったい何を実現しているのでしょう？

MyFirstAppMから見てTimerやLedsがどのようなものであるかは重要ではありません．インターフェースが提供されさえすれば，Timerがどうやってタイマの機能を実現していようがかまわないわけです．このような場合，オブジェクト指向言語ではファクトリ・メソッドなどで目的のインターフェースを継承したインスタンスを生成し，そのメソッド（オーバライドされた仮想関数）を呼び出すという方法をとります．

しかし，TinyOS/nesCが想定しているハードウェア資源では，インスタンスの動的な生成や仮想関数呼び出しのオーバヘッドさえ無視できない処理量となります．ワイヤリングではオブジェクト指向言語と同じように，あるインターフェースを提供する複数のコンポーネントの中から一つを選んで簡単に利用することができます．その一方で，それらのインスタンスの生成や仮想関数の呼び出しはコンパイル時に静的に決定されます．

このようにして，ワイヤリングによってソフトウェアのモジュール性と性能のバランスがうまくとられています．

リスト2　LED点滅モジュール(MyFirstAppM.nc)

```
module MyFirstAppM {
    // 規定部
    provides {
        // 提供インターフェースの宣言
        interface StdControl;
    }
    uses {
        // 利用インターフェースの宣言
        interface Timer;
        interface Leds;
    }
}

implementation {
    // 実装部
    uint8_t counter;

    // StdControlコマンドの実装
    command result_t StdControl.init() {
        counter = 0;
        call Leds.init();
        return SUCCESS;
    }
    command result_t StdControl.start() {
        return call Timer.start(TIMER_REPEAT,100);
    }
    command result_t StdControl.stop() {
        return call Timer.stop();
    }

    // Timerイベントの実装
    event result_t Timer.fired() {
        call Leds.redOff();
        call Leds.greenOff();
        call Leds.yellowOff();

        switch(++counter) {
          case 1:
            call Leds.redOn();
            break;
          case 2:
            call Leds.greenOn();
            break;
          case 3:
            call Leds.yellowOn();
            break;
        }
        counter %= 3;
        return SUCCESS;
    }
}
```

nesCコンパイラのドキュメント生成機能を使って自動的に生成することもできます(図6).

次に，リスト2を見てみましょう．MyFirstAppMはmodule識別子によって宣言されていることからモジュールであることがわかります．

まず，規定部でStdControlを提供することを宣言しており，StdControlのコマンドinit()，start()，stop()を実装部で実装しています．さらに，TimerやLedsといったインターフェースを利用しています．インターフェースを利用する場合，コマンドをcallすることに加えて，利用するインターフェースがイベントを持っているときはそのイベントの実装を行う必要があります．Timerインターフェースがfired()イベントを持つため，これを実装部で実装しています．fired()イベントはTimerが指定された時間経過したときに発生するイベントです．

プログラムは非常に単純です．MyFirstAppM内の変数はcounterだけで，init()でcounterの初期化とLedsの初期化を行い，start()でタイマの開始，stop()でタイマの停止を行っています．そしてfired()イベントの中で赤・緑・黄色のLEDをcounterの値を使って順番に光らせています．

● センサ・データを取得するプログラムに書き換える

次にセンサのデータを取得するように変更してみます．ここでは，光センサと温度センサを搭載したCrossbow社の「MTS310」というセンサ・ボードを使ってデータを取得します．センサ・ボードと無線モジュールの間のインターフェースは拡張用51ピン・コネクタです．このコネクタを介して，無線モジュール上のマイコン(ATmega128L)のA-Dコンバータの入力チャネルに各センサの出力がつながっています．A-Dコンバータは一度で一つのチャネルのデータをディジタル値に変換できます．この変換は非同期に行われ，変換完了時に割り込みが発生します．

また，TinyOSにはMTS310の各センサの値を取得するためのコンポーネントが用意されているので，これを使うことにします．光センサと温度センサはA-Dコンバータのチャネルを共有しているため，これらをまとめたPhotoTempコンポーネントを使います．

リスト3にセンサ値を取得するように変更したトップレベル記述(MyFirstApp.nc)を示します．PhotoTempコンポーネントは，TempStdControlとPhotoStdControlの二つのStdControlインターフェースを提供しています．ただし，これらのStdControlはアプリケーションと同じタイミングで初期化，開始，終了の処理を行うためのものではなく，センサのデータを得たいタイミングに応じてアプリケーションが開始・終了を制御する必要があります．したがって，MainのStdControlにはつながず，MyFirstAppMが利用するStdControlへつなぐようにしています．

**リスト3
センサ・データ取得用トップレベル記述**

PhotoTempコンポーネントを使うことで，チャネルの共有を気にせず光センサや温度センサの結果をそれぞれ得ることができる．

```
configuration MyFirstApp {
    // 規定部
}
implementation {
    // 実装部
    components Main,TimerC,LedsC,MyFirstAppM,PhotoTemp;

    Main.StdControl -> TimerC;
    Main.StdControl -> MyFirstAppM;

    MyFirstAppM.Leds -> LedsC;
    MyFirstAppM.Timer -> TimerC.Timer[unique("Timer")];

    // MyFirstAppMのStdControlインターフェースへのワイヤリング
    MyFirstAppM.TempControl -> PhotoTemp.TempStdControl;
    MyFirstAppM.PhotoControl -> PhotoTemp.PhotoStdControl;

    // MyFirstAppのADCインターフェースへのワイヤリング
    MyFirstAppM.Temperature -> PhotoTemp.ExternalTempADC;
    MyFirstAppM.Light -> PhotoTemp.ExternalPhotoADC;
```

リスト4　センサ・データ取得モジュール

LED点滅モジュール(MyFirstAppM.nc)からの変更点を示す．

```
module MyFirstAppM {
    provides { ... }
    uses {
    ...
        // 温度・光センサの開始・終了を制御するための
        //                          インターフェース
        interface StdControl as TempControl;
        interface StdControl as PhotoControl;

        // 温度・光センサのセンサ値を取得するための
        //                          インターフェース
        interface ADC as Temperature;
        interface ADC as Light;
    }
}

implementation {
    command result_t StdControl.init() {
    ...
        call TempControl.init();
        call PhotoControl.init();
        return SUCCESS;
    }
    command result_t StdControl.stop() {
        call TempControl.stop();
        call PhotoControl.stop();
        return call Timer.stop();
    }
    event result_t Timer.fired() {
        call TempControl.start();   // 温度センサ開始
        call Temperature.getData(); // 温度センサ値の値取得要求
        return SUCCESS;
    }
    // 温度センサ用ADCインターフェースのイベント実装
    async event result_t Temperature.dataReady(uint16_t data) {
        call TempControl.stop();   // 温度センサ停止

        call PhotoControl.start(); // 光センサ開始
        call Light.getData();      // 光センサ値の値取得要求
        return SUCCESS;
    }
    // 光センサ用ADCインターフェースのイベント実装
    async event result_t Light.dataReady(uint16_t data) {
        call PhotoControl.stop(); // 光センサ停止

        // 光センサの値(data)に応じてLED緑を点滅させる
        if(data > 750){
          call Leds.greenOn();
        }else{
          call Leds.greenOff();
        }
        return SUCCESS;
    }
}
```

PhotoTempはこれらに加えて，A-Dコンバータを利用するためのインターフェース(ADC)，ExternalTempADCとExternalPhotoADCの二つを提供します．ADCインターフェースのおもな使いかたは，getDataコマンドでA-Dコンバータに変換を開始させ，変換完了を通知するdataReadyイベントを待つというものです．

MyFirstAppMはLEDを光らせたのと同じようにタイマを使ってセンサのデータを読むタイミングを得ます．温度センサの値を取得するときは温度センサ用のTempControlを開始(start)して，温度センサ用A-Dコンバータである Temperature.getData を呼び出します．Temperature.dataReadyで温度センサの測定結果を保存した後，TempControlを停止(stop)します．次に，光センサ用のPhotoControlを開始してから光センサのLight.getDataを呼んで，Light.dataReadyで光センサの測定結果を得ます．**リスト4**に，LED点滅のMyFirstAppM.ncからの変更部を示します．

● ノードどうしで通信状況を見ながら経路を決定していく

TinyOSはワイヤレス・センサ・ネットワーク用に開発されたOSです．そのため，通信プロトコルの変更に柔軟に対応できるようにネットワーク・コンポーネントは階層化されています．用途やハードウェアに応じて各階層を差し替えることが可能です．実際に多くのプロトコルがTinyOS上で開発されています．

TinyOSのネットワーク階層をOSI参照モデルに当てはめて示したものが**図7**です．XMeshはこの中のネットワーク層に相当し，アドホック・マルチホップ・ネットワーク

において重要な経路制御などを行います．このとき，パケットは図8のように各階層で組み立てられます．

XMeshを用いた場合，センサ・ノードから基地局への経路はすべて図9のように決定されます注11．

注11：執筆時点でTinyOSのCVSに公開されているソースをもとに，図9を作成している．また，説明を簡単にするため，通信状況の評価式など，細かい部分の説明は省いている．

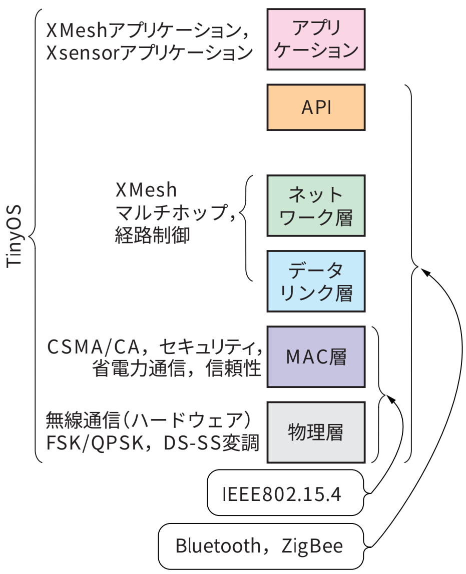


図7　TinyOSのネットワーク階層
TinyOSのネットワーク階層を示した図．本OSはOSI参照モデルの全階層に対応している．

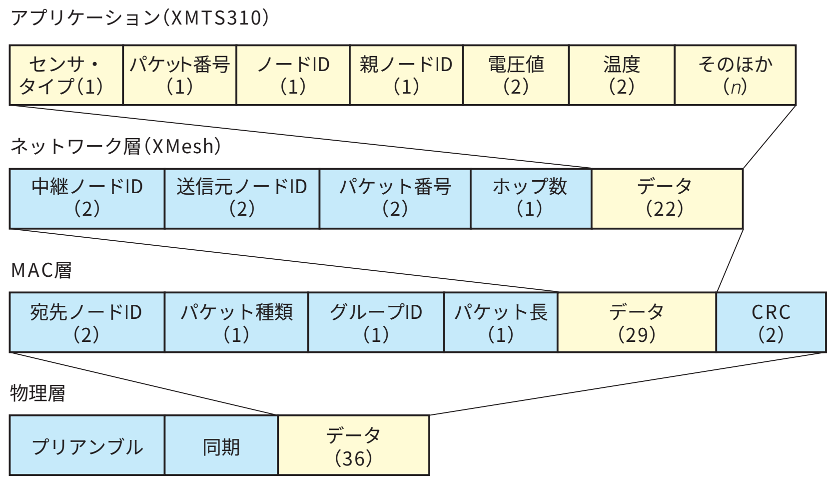


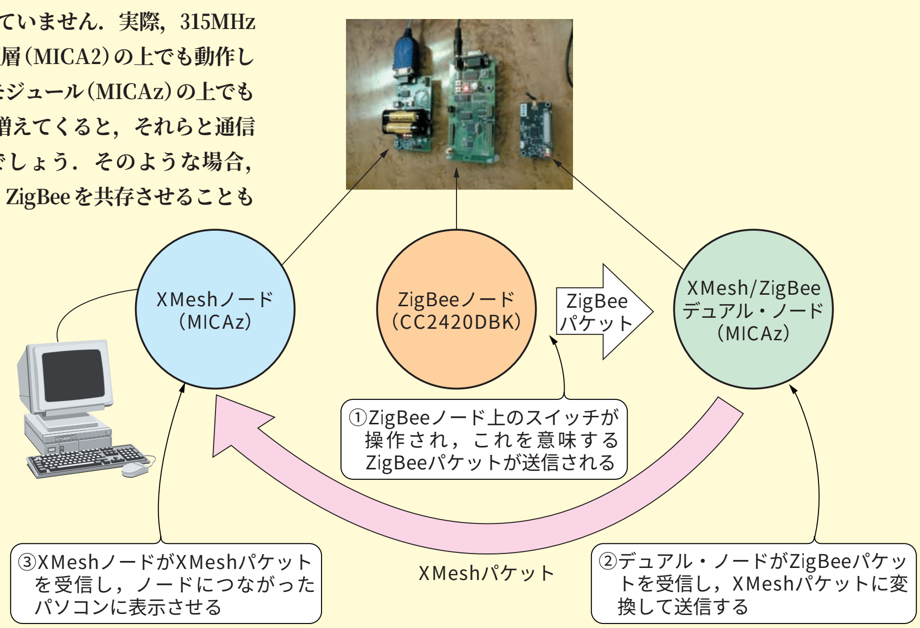
図8　XMeshのパケット構造
パケットは，下位層へ向かって順にヘッダが付けられ，組み立てられていく．

Column　XMeshとZigBee

XMeshとZigBeeの関係についてよく質問を受けます．ZigBeeは物理層とMAC層にIEEE 802.15.4を使い，その上位層にZigBee Allianceで策定したネットワーク・プロトコルを使っています．一方，XMeshはおもにネットワーク層に相当する部分のアドホック・マルチホップ（メッシュ）ネットワークを実現するネットワーク・プロトコルです．

XMeshそのものは物理層まで規定していません．実際，315MHzの微弱電波を使ったTinyOS独自の物理層（MICA2）の上でも動作しますし，IEEE 802.15.4に準拠した無線モジュール（MICAz）の上でも動作します．今後，ZigBee対応製品が増えてくると，それらと通信できることが重視されるようになるでしょう．そのような場合，MICAz上でXmeshを動作させたものとZigBeeを共存させることも可能です．

物理層とMAC層が同じであれば，ZigBeeパケットを受信してXMeshパケットに変換，またはその逆を行うデュアル・ノードをXMesh側に用意することでこれを実現できます（**図A-1**）．つまり，XMeshとZigBeeは用途に応じて使い分けたり，場合によっては共用することもできます．



図A-1
XMeshとZigBeeノードの通信実験図
ZigBeeノードとしてはノルウェーChipcon社のZigBee評価ボードを用いている．この物理スイッチの操作がZigBeeパケットとして送信され，右側のデュアルノードがこれを受信する．デュアル・ノードがXMeshパケットを送信し，左側にあるXMeshノードがこれを受信して，インターフェース・ボードを通じてパソコンに結果を表示する．

まず，各センサ・ノードは一定間隔で経路情報パケットをブロードキャスト（送信）します．経路情報パケットにはほかのセンサ・ノードとの間の送受信状況や基地局からのホップ数，通信コストなどが含まれます．もちろん最初は経路情報は空です．また，すべての送信パケットには順番に番号が付けられます．最初のパケットが0で以後1，2，

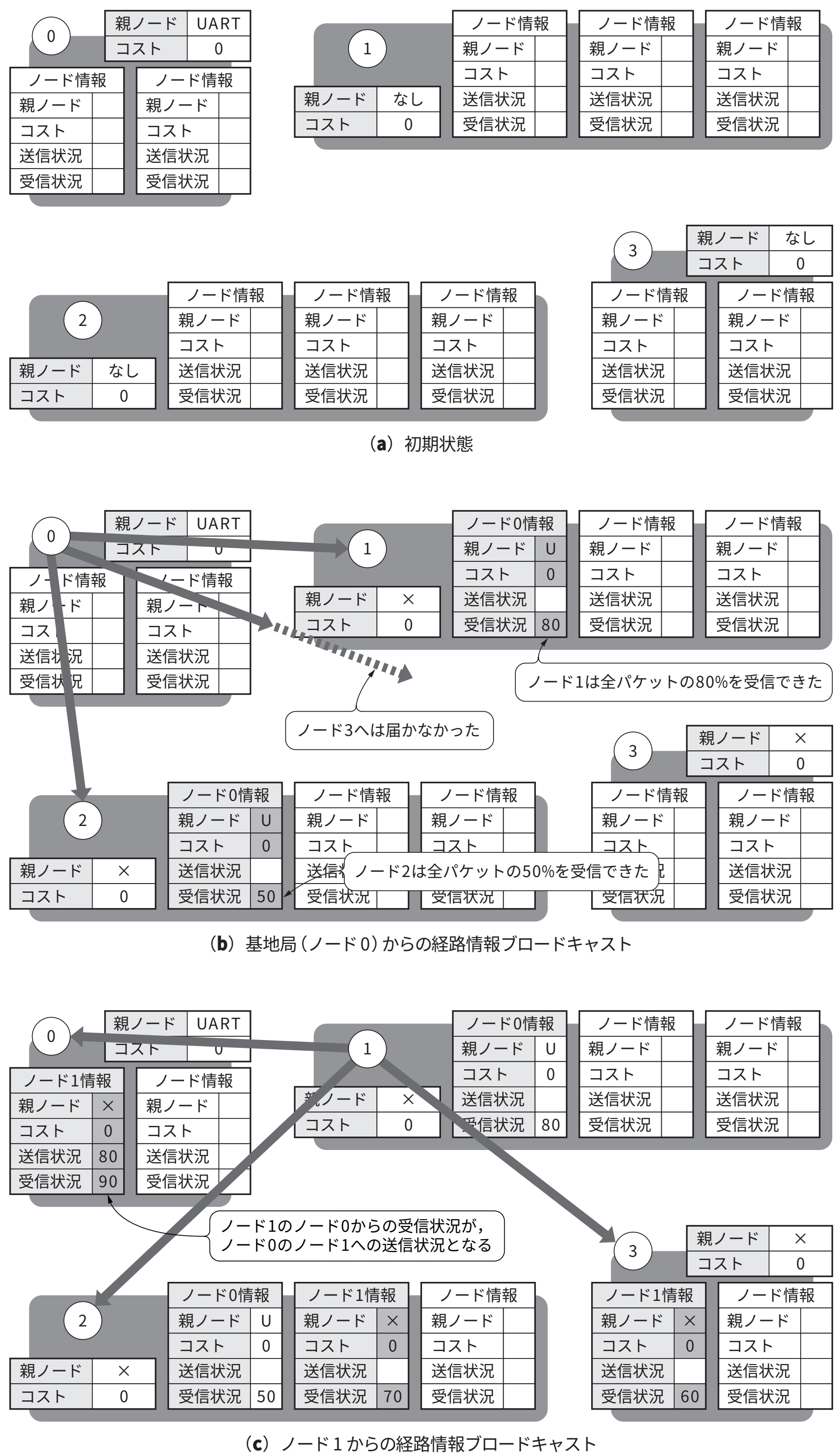
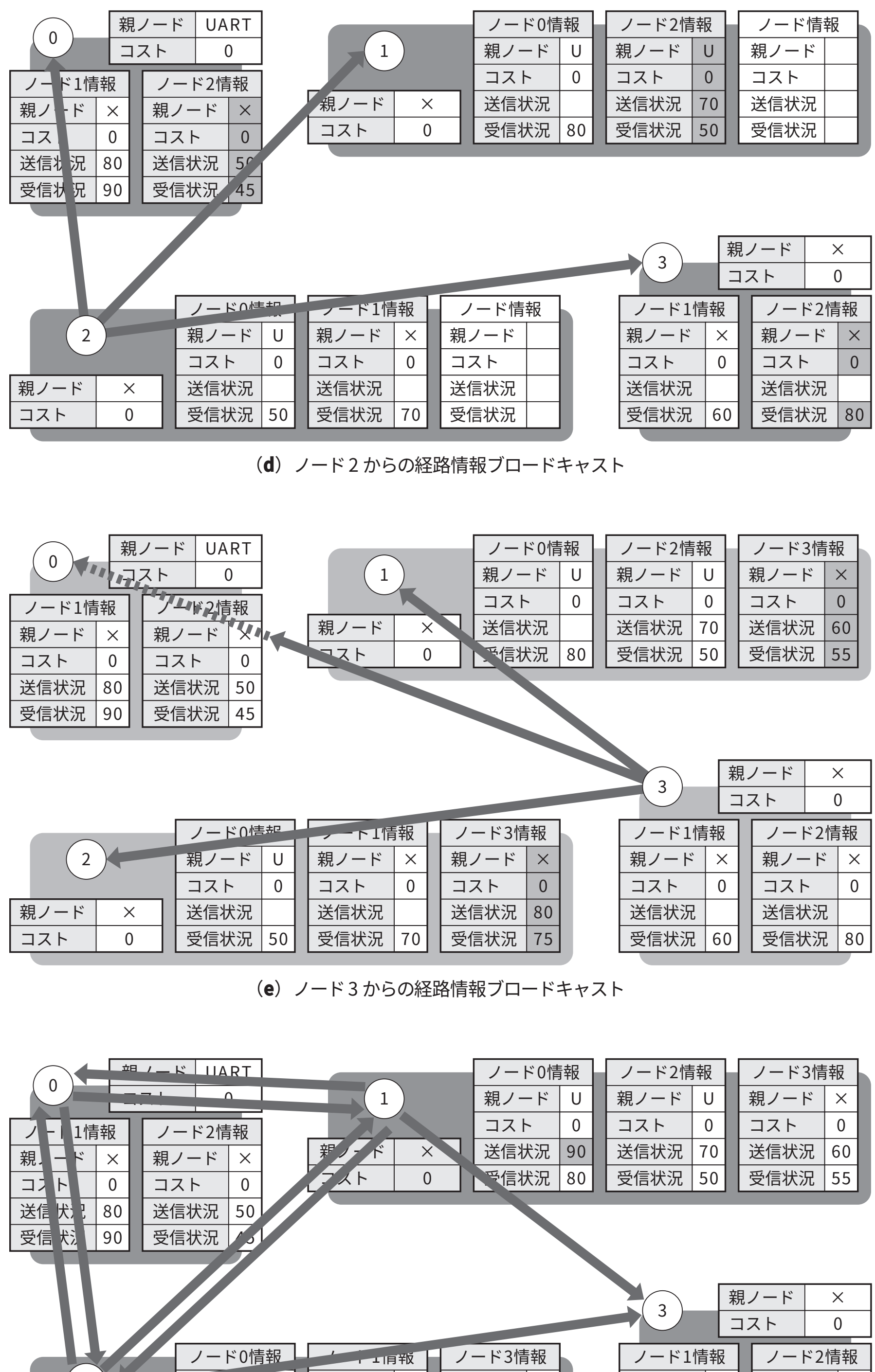


図9
XMeshによる経路決定手順
(a)では，基地局（ノード0）のみ親ノードが決まっている．これは，基地局がシリアル・ポートへデータを出力すると決められているためである．この時点ではほかのノードの親ノードは決まっていない．
(b)では，基地局（ノード0）から経路情報がブロードキャスト（送信）される．現時点で経路情報は空なのでノード0そのものが親ノードとなり，コストが送信される．受信したノードではノード0用の経路情報エントリを作り，受信パケットの番号から受信状況を算出して記憶する．本図では説明を単純にするため，パーセントの値としている．
(c)では，ノード1から経路情報がブロードキャストされる．基地局内のノード1に対する経路情報エントリが完成された．

3，…と付けられていきます．番号付けはネットワーク層，つまりXMeshのパケット・フレーム作成時に行われます．

センサ・ノードは，ほかのセンサ・ノードからパケットを受信すると，そのセンサ・ノードとの通信状況を記憶するための経路情報エントリを作成します．以後，パケットを受信するたびに，受信パケットの番号を調べ，前回受信



（d）ノード2からの経路情報ブロードキャスト

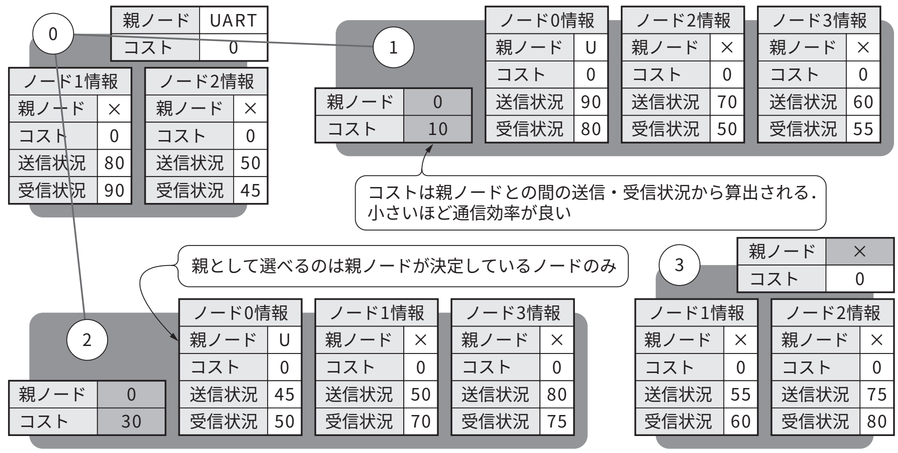
（e）ノード3からの経路情報ブロードキャスト

（f）ノード0，1，2からの経路情報ブロードキャスト

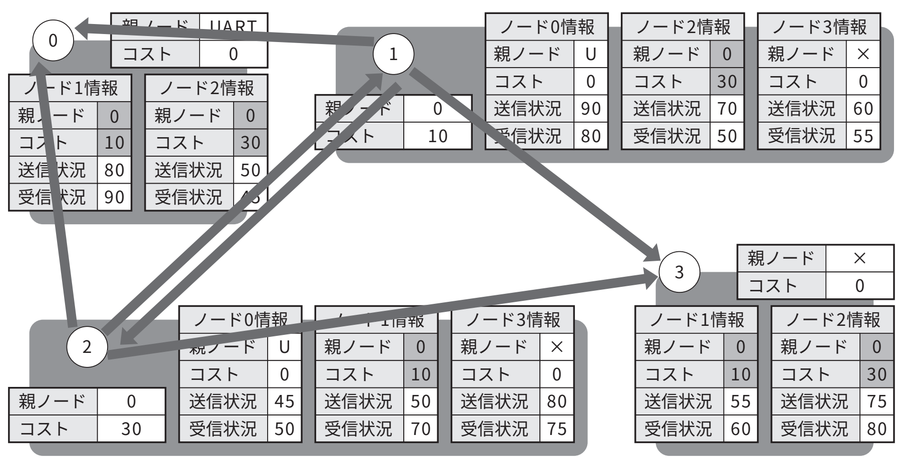
図9
XMeshによる経路決定手順（つづき）

（d）では，ノード2から経路情報がブロードキャストされる．基地局とノード1の中のノード2に対する経路情報エントリが完成する．
（e）では，ノード3から経路情報がブロードキャストされ，ノード1と2の中のノード3に対する経路情報エントリが完成する．
（f）で，再度ノード0～2から経路情報がブロードキャストされる．経路情報は各センサ・ノードが一定時間ごとにブロードキャストする．当然，状況が変われば最新の状況に更新される．

時の番号と比較することにより，受信できなかったパケットの数を調べます．これをもとにそのセンサ・ノードからの受信状況を評価します．経路情報エントリは最大16，すなわち16のセンサ・ノードとの間の通信状況を記憶するようになっています．この数を超える場合，もっとも通信状況の悪いセンサ・ノードの経路情報エントリが削除され，



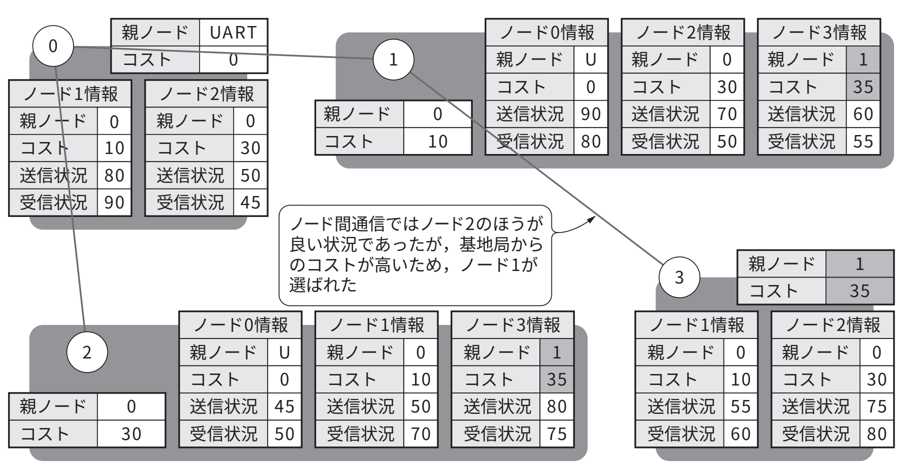
(**g**) 親の決定（ノード1，2）



(**h**) ノード1，2からの経路情報ブロードキャスト

図9
XMeshによる経路決定手順
(つづき)

(g)は，経路情報のブロードキャストをある決められた回数行った後，各ノードが経路情報エントリ内から親ノードを選んだときのようす．親ノードが決定していないノードは対象外となるため，この例ではノード1，2ともに基地局（ノード0）を選択する．親が決まれば親のコストに，親との間の送受信状況から計算された親との間のコストを加えた値を自身の基地局からのコストとして記憶する．
親とコストが決まった後，ノード1，2からの経路情報ブロードキャストを(h)で行っている．これでノード3でも親ノードを選択できる条件がそろう．
(i)において，ノード3で親が決定される．ノード間の通信状況はノード2のほうが良いのだが，基地局からのコストを加えて評価するため，この場合はノード1が選択された．



(**i**) 親の決定（ノード3）

リスト5　データ送信用トップレベル記述

```
ncludes MyFirstAppMsg;

configuration MyFirstApp {
    // 規定部
}
implementation {
    // 実装部
    components Main,TimerC,LedsC,MyFirstAppM,PhotoTemp,
    EWMAMultiHopRouter as MultiHopM,  // マルチホップ用
    GenericCommPromiscuous as Comm;
                                // プロミスキュアス受信用

    Main.StdControl -> TimerC;
    Main.StdControl -> MyFirstAppM;
    Main.StdControl -> MultiHopM;
    Main.StdControl -> Comm;

    MyFirstAppM.Leds -> LedsC;
    MyFirstAppM.Timer -> TimerC.Timer[unique("Timer")];

    // MyFirstAppMのStdControlインターフェースへの
    //                                       ワイヤリング
    MyFirstAppM.TempControl -> PhotoTemp.TempStdControl;
    MyFirstAppM.PhotoControl ->
                              PhotoTemp.PhotoStdControl;

    // MyFirstAppのADCインターフェースへのワイヤリング
    MyFirstAppM.Temperature ->
                              PhotoTemp.ExternalTempADC;
    MyFirstAppM.Light -> PhotoTemp.ExternalPhotoADC;

    // マルチホップ・メッセージを使うためAM_XMULTIHOP_MSGを
    //                                             指定する
    MyFirstAppM.Send ->
                      MultiHopM.Send[AM_XMULTIHOP_MSG];
    MultiHopM.ReceiveMsg[AM_XMULTIHOP_MSG] ->
                     Comm.ReceiveMsg[AM_XMULTIHOP_MSG];
}
```

リスト6　データ送信用モジュール

センサ・データ取得モジュール(リスト4)からの変更点を示す.

```
module MyFirstAppM {
    provides { ... }
    uses {
    ...
        // センサ・データ送信用インターフェース
        interface Send;
    }
}

implementation {
    // 実装部
    TOS_Msg     gMsgBuffer;    // 送信パケット用バッファ
    uint16_t    temperature;
    uint16_t    light;

    // センサ・データ送信タスク
    task void sendTask() {
        MyFirstAppMsg   *data;
        uint16_t        len;

        call Leds.yellowOn();

        // 送信パケットのセット
        data = (MyFirstAppMsg*)call Send.getBuffer
                               (&gMsgBuffer,&len);
        data->node_id = TOS_LOCAL_ADDRESS;
                  // 自分のノードIDはこの変数に保持されている
        data->light = light;
        data->temperature = temperature;

        if(call Send.send(&gMsgBuffer,sizeof
                                (MyFirstAppMsg)) !
                                     = SUCCESS){
          // 送信失敗
          call Leds.yellowOff();
        }
    }
    ...
    // 光センサ用ADCインターフェースのイベント実装
    async event result_t Light.dataReady(uint16_t data) {
        call PhotoControl.stop();  // 光センサ停止
        light = data;

        // sendTaskタスクをシステムのタスク・キューに登録する
        // この時点でsendTaskが呼び出されるわけではない
        post sendTask();

        return SUCCESS;
    }
    // 送信完了イベント
    event result_t Send.sendDone(TOS_MsgPtr msg,
                              result_t success) {
        call Leds.yellowOff();
        return SUCCESS;
    }
}
```

新しいエントリが使われます.

こうして得られた情報は経路情報パケットに入れられてブロードキャストされます.パケットの大きさの制限から16エントリすべてを一度に送信できず,いくつかに分割して送信します.

経路情報パケットを受信したセンサ・ノードは,その中に自分から相手ノードへの経路情報がないかを調べ,あった場合はその中の受信状況を自分からそのセンサ・ノードへの送信状況として使います.このようにして得られた受信状況,送信状況をもとにしてもっとも通信状況の良いセンサ・ノードを親ノードとして決定します(ただし,自分の親ノードが決定されていなければ,ほかのノードの親にはなれない).また,親ノードの候補のそれぞれに対して,基地局(すべての最終的な親となる)からの通信コストを積算していき,それを親ノードの決定時の評価に使います.こうすることにより,経路の選択は2ノード間の通信状況からだけでなく,ネットワーク全体の通信コストが低くなるように決定されます.

親ノードが決まると,各センサ・ノードでは自分宛のパケットが届けばそれを親ノードへ転送し,それ以外のパケットは経路制御のための情報として使うためにネットワーク層まで受信した後,破棄されます.このようにして各セ

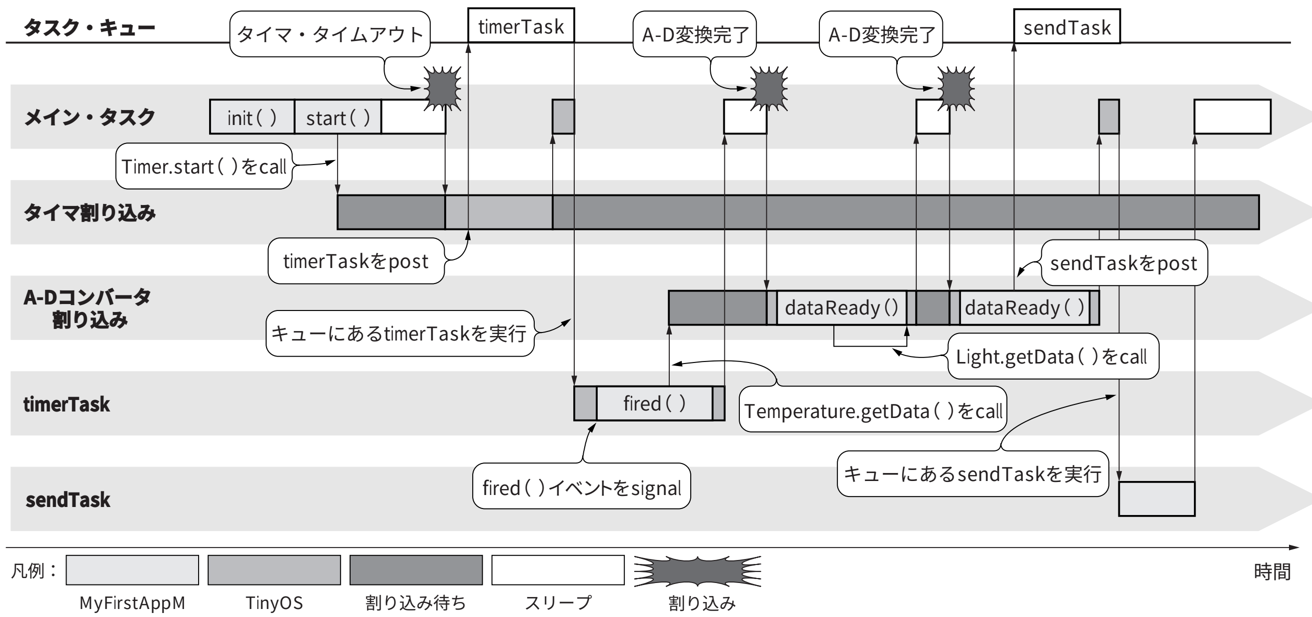


図10　MyFirstApp でのタスク・割り込み実行シーケンス
初期化処理を除けば，「割り込みの発生→タスクのpost→タスクの実行・完了→スリープ→割り込みの発生，…」を繰り返していることがわかる．

ンサ・ノードが親ノードを持ち，最終的に基地局へとデータが届けられる経路が完成します．

● センサ・データ送信用アプリケーションを作る

では，実際にXMeshを使ってセンサのデータを送信してみましょう．**リスト5**と**リスト6**にトップレベル（MyFirstApp.nc）とモジュール部（MyFirstAppM.nc）を示します[注12]．

センサのデータを送信するために使うインターフェースはSendです．手順は，getBufferコマンドでTinyOSパケット中のデータ部分のアドレスを得て，そこにデータを書き込み，sendコマンドでパケットを送信します．送信処理は非同期で行われ，sendDoneイベントで送信が完了したことが通知されます．Sendインターフェースを提供するXMeshのコンポーネントはEWMAMultiHopRouterです．EWMAMultiHopRouterはStdControlを提供し，Mainコンポーネントへつなぐ必要があります．これは先に述べたように経路情報パケットの送信をバックグラウンドで行うためです．

パラメータ付きのSendインターフェースが提供されています．パラメータは送信するパケットの種類を指定するためのものです．マルチホップ・パケットを使うので，パラメータとしてAM_XMULTIHOP_MSGを指定します（これ以外にもインターフェースが提供されているが，今回のアプリケーションでは使わないため，ワイヤリングしない）．

EWMAMultiHopRouterはReceiveMsgインターフェースを利用するので，これを提供するコンポーネントをワイヤリングしなければなりません．ReceiveMsgインターフェースを提供するコンポーネントとしてGenericCommPromiscuousを使います．GenericCommコンポーネントと違ってプロミスキュアス受信を行うコンポーネントです．プロミスキュアス受信とは，自分以外が宛先のパケットも受信することであり，先に述べたようにXMeshがほかのノードとの通信状況の測定に使います．GenericCommPromiscuousもStdControlを提供しているのでMainにつないでいます．

これでデータを送信するためのコンポーネントのワイヤリングはできました．後は，温度センサと光センサのデータが取得できた段階で，実際にsendコマンドを呼び出してデータを送信するだけです．

● タスクは発生→実行→スリープ→割り込みの繰り返し

データを送信するタイミングは，光センサのdataReady()イベントでとることができます．ただし，ここで注意すべき点は，dataReady()イベントはasync指定子が

注12：リスト5およびリスト6は，データ競合関連の警告（warning）が出るが，動作する．CD-ROM収録のソースには競合回避処理も入っているので，参考にしていただきたい．

付いており，割り込みハンドラ内から呼ばれているということです．そのため，dataReady()イベントを実装する際にはデータ競合が起こりうることに留意しておく必要があり，時間のかかる処理を行わないようにするべきです．そこで，データの送信処理そのものはdataReady()内では行わずにタスクで行うようにし，dataReady()ではタスクをpostするだけにします．postはタスクをタスク・キューに登録するために使います．

MyFirstAppにおけるタスク，割り込みの実行状況を**図10**に示します．まず，メイン・タスク(MyFirstAppM)のstart()でタイマが開始されます．その後，タスク・キューが空なのですぐにスリープ状態に入ります．タイマがタイムアウトした時点でタイマ割り込みが発生します．Timer.fired()にasyncが付いていないのは，タイマの割り込みハンドラからこのイベントが呼ばれているわけではなく，このイベントをsignalする処理そのものはタスクとして割り込みハンドラ内から登録されているからです．

割り込みによって再びタスク・キュー内のタスクを実行しにいき，そこで先に登録されたTimer.fired()をsignalするタスクが実行されます．Timer.fired()イベントが呼ばれ，その中で今度は温度センサのgetData()が呼ばれます．これでタスクがなくなりスリープに入りますが，次にA-Dコンバータの変換完了による割り込みがかかります．これは直接dataReady()イベントをsignalし，この中で光センサのgetData()が呼ばれ，その後，同じようにdataReady()イベントが発生します．ここで，センサのデータを送信するsendTask()をpostしています．割り込み処理が終わったらキューに入っているsendTask()が実行されます．その後，スリープして再びタイマのタイムアウト割り込みがかかり，以後同じ処理が繰り返されていきます．

タイマの間隔によっては，sendTask()にタイマのタイムアウトが割り込むこともありえます．このとき，イベント呼び出しはタスクで行われるため，かならずsendTask()が終わった後でTimer.fired()が呼ばれます．

注13：以後，先頭に'/'を付けていない場合はすべてCygwin内のパスの/opt/tinyos-1.x以下の相対パスとする．なお，xlistenは，米国OSTG(Open Source Technology Group)社が運営しているオープンソース・ソフトウェアの開発サイトSourceForge内のTinyOSプロジェクト(http://sourceforge.net/cvs/?group_id=28656)のCVSから取得することも可能．

注14：xlistenのソースを変更して作成したプログラムを再配布する場合は，ライセンスに従うように注意すること．

● パソコンでセンサのデータを受信する

ここまでで，センサのデータをXMesh経由で基地局まで送信するようにしました．次は，基地局からパソコンへデータを送り，それをパソコン側で取得できるようにします．今回のシステムでは，親ノードのIDはシリアル・ポート(UART)へ出力するように設定されているため，基地局からシリアル・ポートへデータの送信が行われています．パソコン側でシリアル・ポートからデータを読み出せばセンサのデータを取り込むことができます．

これを行うために，xlistenというプログラムを使います．参考文献(4)，(5)に従って必要なソフトウェアをインストールすると，Windows上で動作するUNIX環境であるCygwinもインストールされます．このCygwin内のパス/opt/tinyos-1.x/contrib/xbow/tools/src/xlistenにxlistenがあります[注13]．この下のboardsというディレクトリの下に各センサ・データの構造に応じてパケットの処理を行うためのソースが集められています．独自のセンサ・データを処理したい場合，ここにあるソースを参考にして新しいセンサ用のエントリを追加するとよいでしょう[注14]．

また，単にパケットの内容を表示したいだけならxlistenの-rオプションが使えます．**図8**のTinyOSのパケットをそのまま見ることができますが，以下の2点に注意してください．

- ヘッダとして0x7E，0x42の2バイトが付いている
- 0x7E，0x7Dというデータがパケット中にあった場合，エスケープ処理されて表示される．具体的には，まず0x7Dを表示し，その次に実際のデータを0x20でXORしたものを表示する．例えば，0x7Dであれば，0x7D，0x5Dというように2バイトで表示される

● 三つのデバッグ方法

nesCのデバッグにはおもに以下の三つの方法があります．

1) LEDを用いたデバッグ

Ledsコンポーネントを使って，イベント発生時やある条件が成り立ったときなどに，それがわかるようにLEDを点灯させるというものです．無線モジュール(MOTE)に搭載されている三つのLEDをうまく使って情報を表現するくふうが必要ですが，非常に簡単に使える方法です．またLedsにはsetというコマンドがあり，7(3ビット)までの数字をLEDで表すことも可能です．

2) シリアル・ポートを用いたデバッグ

Column　ワイヤレス・センサ・ネットワークのセキュリティ

無線通信においてセキュリティは重要です．ワイヤレス・センサ・ネットワークにおいても重要なセンサ・データを傍受されたり，偽のパケットを流してシステムを乗っ取られるといったことは大きな問題となりえます．

TinyOSにはソフトウェア・レベルで送信データを暗号化するためのコンポーネント「TinySec」が用意されています．TinySecはデータリンク層で暗号化を行います．

一方，IEEE 802.15.4にはAES-128（かぎの長さが128ビット）による暗号化が規定されており，MICAzで使われているRFトランシーバIC「CC2420（Chipcon社製）」はこれに対応しています．ただし，本記事執筆時点ではCC2420の暗号化機能を制御するインターフェースはまだなく，CC2420ドライバに機能を追加する必要があるようです．

シリアル・ポートを用いたデバッグを行うためには，無線モジュール（MOTE）とパソコン上のターミナル・ソフトウェアをつなぐためのインターフェース・ボードが複数必要になりますが，LEDよりも多くの情報を出力することができます．

まず，Cygwinウィンドウでcontrib/xbow/tos/interfaceからtos/interfacesへSOdebug.hをコピーします．デバッグしたいソースで**リスト7**のようにいくつかのおまじないを行った後，デバッグ情報を出力したい場面でSODbg()を呼び出します．後は，ターミナル・ソフトウェアで通信速度を57600bps，データ・ビットを8，パリティなし，ストップ・ビットを1，フロー制御なしとして接続すれば，無線モジュールからの出力を確認することができます．

3）JTAGを用いたデバッグ

JTAGを用いると，デバッガを使ってソース・レベルのデバッグが行えるため，もっとも正確に検証できます(6)．ただし，通信など，タイミング的にシビアな部分のデバッグはプログラムを止めると正常に動作しなくなるため，いろいろなくふうを施す必要があります．

また，nesCの記法がわからない場合は，nesCコンパイラが生成したCソースを調べるという方法があります．通常はアプリケーション・ディレクトリの下に作られるbuild/<platform>/app.cというファイルがこれに該当します．nesCに慣れないうちはこのCソースを見ながら学ぶとよいでしょう．

リスト7　SODbgを使うための準備

```
// ラベルの定義と，SOdebug.hのinclude
implementation {
    #define SODBGON 1
    #include <SOdebug.h>
    ...
}

// SOdebugモジュールの初期化
command result_t StdControl.init() {
    init_debug();
    ...
}

// SODbg呼び出し
.. {
    ...
    SODbg(SODGBON, "value=%i¥n",value);
    ...
}
```


参考・引用*文献

(1) クロスボーのホームページ，http://www.spp.co.jp/xbow/motemica.html（MOTEに関する日本語資料）．
(2) Crossbow Technology社のMOTEに関するホームページ，http://www.xbow.com/Products/productsdetails.aspx?sid=3
(3) UC BerkeleyのTinyOSに関するホームページ，http://www.tinyos.net/
(4) Crossbow Technology；Getting Started Guide（http://www.xbow.com/Support/Support_pdf_files/Getting_Started_Guide.pdf），April 2005.
(5) クロスボー；MOTE-Wireless Sensor Netoworks Getting Started Guide日本語ガイド（http://www.spp.co.jp/xbow/gsgj117.pdf），2004年8月.
(6) クロスボーのMOTEサポート・ページ，http://blog.livedoor.jp/sssj/archives/24795080.html
(7) ATmega128Lのデータシート，http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
(8) CC2420のデータシート，http://www.chipcon.com/files/CC2420_Data_Sheet_1_2.pdf

みやもと・さとる
住友精密工業（株）MEMS事業開発室システム開発グループ

<筆者プロフィール>
宮本 哲．電機メーカでPDAや携帯電話のソフトウェア開発業務を経て，2005年3月に転職．現在は住友精密工業にてCrossbow社のMOTEを使ったワイヤレス・センサ・ネットワークの開発に従事．

4